ANL-21/17

CodeFlow: A Code Generation System for Flash-X Orchestration Runtime

Mathematics and Computer Science Division

ANL-21/77

CodeFlow: A Code Generation System for Flash-X Orchestration Runtime

prepared by

Johann Rudi*, Jared O'Neal*, Mohamed Wahib†, Anshu Dubey*

* Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439. Email: {jrudi,joneal,adubey}@anl.gov

† National Institute of Advanced Industrial Science and Technology, Japan. Email: mohamed.attia@aist.go.jp

Mathematics and Computer Science Division

4/12/2021

CodeFlow:
A Code Generation System for Flash-X Orchestration Runtime

Johann Rudi[1], Jared O'Neal[1], Mohamed Wahib[2], and Anshu Dubey[1]

[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]National Institute of Advanced Industrial Science and Technology, Japan

April 12, 2021

Abstract

We propose the CodeFlow toolchain for Flash-X that realizes the "recipe-to-source" code transformation for Flash-X simulations and that is necessary to achive performance portability. We design a high-level language to express operations of simulations in so-called recipes, which are given as input to the toolchain. The tools of the CodeFlow pipeline include code transformation with tree-based source code representation techniques and code orchestration and generation based on control flow graphs. The generated source code utilizes a new runtime, developed for Flash-X, that orchestrates dynamic and asynchronous data movement and task execution. The functionality of CodeFlow is demonstrated using a hydrodynamic problem with a strong shock.

1 Introduction

We present discrete tools that are linked with each other to arrive at a code generation pipeline for a particular scientific applications, Flash-X. To this end, we propose the *CodeFlow* toolchain for Flash-X. The tools of the toolchain build on two main concepts, namely code transformation with tree-based source code representation techniques and code orchestration and generation with graph algorithms for the control flow of Flash-X simulations. Flash-X is a new code derived from FLASH [5]. It builds up the architecture from ground up and it performs large-scale astrophysics multiphysics simulations. One crucial addition to Flash-X is the new orchestration runtime [10]. We aim to deploy CodeFlow to generate source code that creates the interface between static Flash-X code (e.g, of physics units) and the runtime, which features dynamic asynchronous task execution and data movement.

We introduce multiple levels of abstraction to source code of Flash-X simulations, which arise from utilizing CodeFlow, in order to achieve performance portability on heterogeneous hardware architectures of compute nodes of leadership-class supercomputers. Currently, these nodes typically consist of one or multiple CPUs and one or several GPUs; however, architectures are likely to change in the future and our toolchain-based approach aims to adapt to unknown future architectures (e.g., accelerators and FPGAs). The adaptability to unknown architectures is possible by designing tools that are flexible and that users can modify according to the requirements of applications or computing platforms. CodeFlow aims to accomplish this by allowing source code generation in any programming language and by transparency of the code generation process to users.

Background As scientific applications such as Flash-X become more realistic, the complexities of their implementation codes increase dramatically. Simultaneously, the hardware of parallel high-performance systems transitions to increasingly heterogeneous architectures with CPUs and accelerators (e.g., GPUs) having different computing throughput and memory bandwidth, requiring additional programming efforts. This challenging landscape of complex scientific application codes and intricate hardware utilization demands adaptable and maintainable software that is capable of maintaining good parallel performance, which is often summarized as performance portability.

High-level abstractions have been introduced to hide the platform complexity from the scientific components of the software. Most currently existing tools, for instance Kokkos [6], Raja [1], AMReX [11], and STELLA/GridTools [7], rely on template metaprogramming in C++ to implement the abstractions. This makes these tools less attractive for applications written in other languages, for instance, Fortran in the case of Flash-X. Additionally, the tools require users to fully embrace one abstraction tool and to completely rely on it for performance. This prohibits the use of expert knowledge by users about their application or a specific target platform.

The Flash-X software was recently augmented by a runtime [10], which has been developed by the authors and other collaborators. The runtime exposes the underlying parallelism in heterogeneous hardware nodes in such a way that computational code can make efficient use of the hardware's resources. As part of this, the runtime is responsible for the orchestration of both data movement and task execution with the goal of maximizing asynchronous operations. This development opened the opportunity for creating automated code generation tools that create interface code between static physics code of Flash-X and the runtime.

Contributions We propose the CodeFlow toolchain, where users write a recipe of a Flash-X simulation using a domain specific language (DSL). The DSL is designed to express a control flow of operations. The syntax encodes dependencies between the operations, and it is similar to the functional syntax of TensorFlow/Keras. CodeFlow generates a control flow graph associated with a recipe, which takes the form of a directed acyclic graph. It analyzes and transforms the graph in order to be able to extract sets of subroutines performed locally on a device (e.g., CPU or GPU) and to infer data movement between devices of a heterogeneous compute node.

The control flow graph is transformed into a hierarchical graph, which enables parsing of source code that interfaces with the runtime by providing the implementation of data packets, whose movements the runtime will oversee, and by autogenerating device-specific code for subroutines that are launched by the runtime.

Additionally, CodeFlow aims to be transparent to users, and it aims to provide control mechanisms for experienced users who want to perform optimizations themselves or when pure automation is not sufficient for achieving performance targets. In the latter case, users should be able to intervene in intermediate results of CodeFlow in order to incorporate their expert knowledge about the application and target platform.

2 Code transformation with source trees and abstract syntax trees

This section presents two code transformation tools. We propose a new technique in Section 2.1 that exploits tree topologies for source code generation. Our approach is based on a simplification of abstract syntax trees, which are described in Section 2.2.

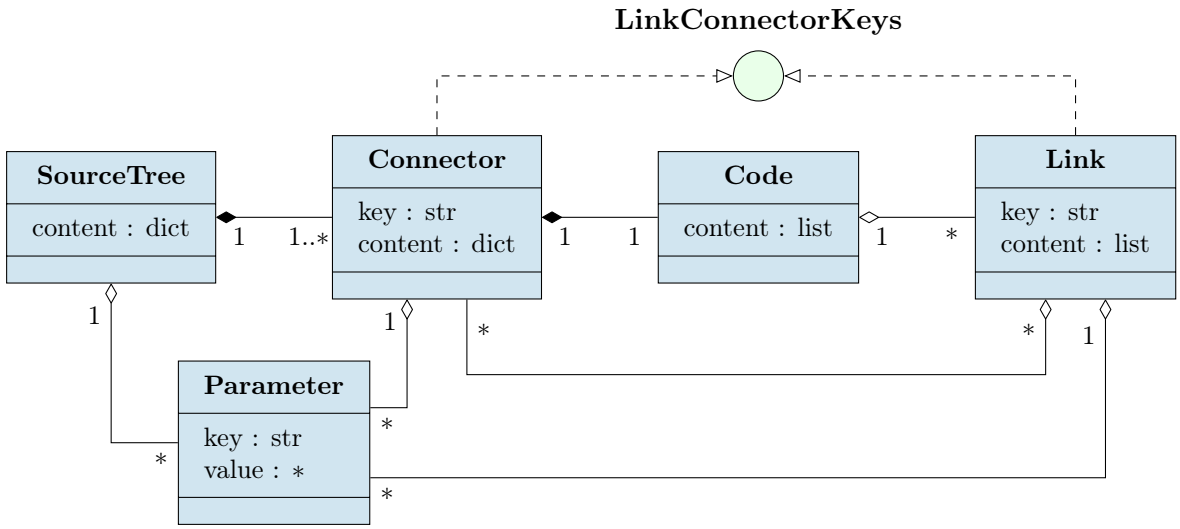

Figure 1: UML diagram of parametrized source trees. A SourceTree is composed of one or multiple Connectors that contain one Code object; Code contains lines of source code or provide a Link. Links and Connectors are associated by matching keys. Parameters can exists at the SourceTree level and within Connectors.

2.1 Parametrized source trees

We propose code transformation with *parametrized source trees (PSTs)*. PSTs are trees of source code that can be assembled by automated tools based on predefined sets of templates. Each template contains lines of source code and also placeholders for including additional code. These placeholders, which we call links, represent nodes in the source tree topology. To complement the links, templates also specify connectors, which, in turn, enable each template to be included at any level of the PST.

The PST design is illustrated in Figure 1, and while it is based on just three simple concepts of source code, links, and connectors, it allows for a great versatility in code generation. The source trees are in addition parametrized (Parameter class in Figure 1) such that the templates can take into account context information. The parameters are passed down the tree hierarchy such that parameter definitions at higher-levels can be used at lower levels of the tree. In order to access a parameter, the source code simply contains the parameter's unique key, and a substitution will take place when the assembled PST is parsed.

The correctness of a PST is verified by checking its structure and by asserting the existence of all parameter definitions. Additionally, it is always possible for users to inspect intermediate PSTs at any stage of the assembly. When parsing a PST, the user can request a verbose output, where commented lines are inserted around lines of code in order to refer to the template that is responsible for these lines.

We illustrate the usage of PSTs with a simple example of an AXPY operation, shown in the templates of Listings 1 to 4 using the JSON file format. Listing 1 implements the main function and provides links for function definitions before the main function (`_link:setup`) and code inside the main function (`_link:execute`). Listings 2 and 3 implement two alternative versions for performing the AXPY operation, and either one or the other is connected to the links in Listing 1. Listing 2 is the template for a GPU code and Listing 3 is CPU code. Finally, Listing 4 isolates the AXPY arithmetic in a singe template, which is possible due to the parametrization of the preceding templates. The example also illustrates the use of parameters to change the variable types of the floating point arrays (`float` or `double`) via `_param:axyType`; and it shows the possibility of substituting constants, such as

```
{
  "_connector:main": {
    "_param:axyFunction": "saxpy",
    "_param:axyType":     "float",
    "_param:size":        3200000,
    "_param:a": "a",
    "_param:x": "x",
    "_param:y": "y",
    "_code": [
      {
        "_param:indent": 0,
        "_link:setup": []
      },
      "int main(void)",
      "{",
      "  /* ... initialize ... */",
      {
        "_param:indent": 1,
        "_link:execute": []
      },
      "  /* ... finalize ... */",
      "}"
    ]
  }
}
```

Listing 1: PST example of AXPY: Template for main file.

`_param:size`, directly into parsed source.

The main motivation behind our parametrized source trees is to create a lightweight alternative to abstract syntax trees, which significantly reduces implementation efforts for users. However, transformation of abstract syntax trees can be advantageous in some cases of code transformation, which we describe next.

2.2 Abstract syntax trees

Abstract syntax trees (ASTs) [4, 9] are rich in the context information they can provide for code transformation. This is especially important if an analysis of source code has to be performed before code transformation can take place. We aim to use the powerful concept of source-to-source transpilation via ASTs [2, 3] in order to analyse and transform low-level computational kernels, which are typically nested loops containing computationally significant floating point arithmetic. These kernels are transformed and optimized to adapt the code to different hardware architectures.

The main advantage of AST-based transpilation is the ability to ingest code directly without needing the intervention by users to incorporate a code transformator, such as PST, into their workflow. However, this comes at higher development efforts for controlling AST-based code transformation. In our preliminary work the AST transpilation approach has demonstrated to be effective for low-level computational kernels because of their compact source codes. In addition to code transformation, the AST can be essential in analysing source code, for example, extracting function declarations and classification of function arguments.

```
{
  "_connector:setup": {
    "_param:idx": "i",
    "_code": [
      "__global__",
      "void _param:axyFunction(_param:axyType _param:a,",
      "                        _param:axyType *_param:x,",
      "                        _param:axyType *_param:y)",
      "{",
      "  int _param:idx = blockIdx.x*blockDim.x + threadIdx.x;",
      {
        "_param:indent": 1,
        "_link:execute": []
      },
      "}"
    ]
  },
  "_connector:execute": {
    "_param:threadBlockSize": 320,
    "_code": [
      "_param:axpFunction",
      "    <<<_param:size/_param:threadBlockSize, _param:threadBlockSize>>>",
      "    (_param:a, _param:x, _param:y);"
    ]
  }
}
```

Listing 2: PST example of AXPY: Template for GPU kernel function and kernel execution.

```
{
  "_connector:execute": {
    "_param:idx": "i",
    "_code": [
      "for (int _param:idx=0; _param:idx<_param:size; _param:idx++) {",
      {
        "_param:indent": 1,
        "_link:execute": []
      },
      "}"
    ]
  }
}
```

Listing 3: PST example of AXPY: Template for CPU loop.

```
{
  "_connector:execute": {
    "_code": [
      "_param:y[_param:idx] = _param:a*_param:x[_param:idx] + _param:y[_param:idx];"
    ]
  }
}
```

Listing 4: PST example of AXPY: Template for arithmetic operation.

```
aR = Action(routine='function_R')()
aS = Action(routine='function_S')(aR)
aT = Action(routine='function_T')(aS)

aX = Action(routine='function_X')(aS)
aY = Action(routine='function_Y')(aX)
aZ = Action(routine='function_Z')([aT,aY])
```

Listing 5: Example of Pipeline pattern.

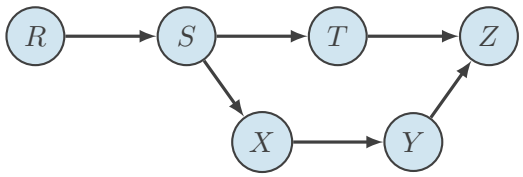

Figure 2: Example control flow graph of Pipeline pattern, corresponding to Listing 5.

3 Code orchestration with hierarchical control flow graphs

The code transformation systems of Section 2 can be utilized directly by users as well as by automated tools. This section describes our code generation approach with CodeFlow, which uses our PSTs from Section 2.1 for code transformation.

We design a new high-level DSL that allows users to express control flows of applications in form of concise recipes (see Section 3.2). The DSL is based on the Python language and it follows a so called "define and run" principle. This principle entails that a recipe describes the composition of building blocks in form of subroutines, called actions, and their dependencies. The syntax for recipes is derived from the patterns in Section 3.1. Recipes are processed into control flow graphs, which subsequently are analyzed and transformed as detailed in Section 3.3. The resulting transformed control flow graph can be traversed to assemble source code based on PSTs.

3.1 Patterns

In this section, we identify several patterns that need to be supported by our recipe DSL for Flash-X. In addition, it is possible to further extend the syntax of the DSL in the future, if we need to support new patterns for new classes of applications beyond Flash-X.

The important patterns for expressing operations of Flash-X at the recipe level are based on the following main concepts: (i) dependencies between operations; (ii) concurrent data items that a single operation is executed on; (iii) concurrent operations that are executed on a single data item; (iv) mapping operations to hardware of a compute node.

Pattern: Pipeline Pipelines express the execution order of actions and dependencies of actions to one another. *Actions* are defined as operations that are performed on blocks of the computational mesh. Data flows through a Pipeline in a concurrent fashion. The Pipeline pattern is fundamental for the realization of the control flow graphs that are presented in Section 3.3. We illustrate this pattern with an example in Listing 5 and show the corresponding control flow graph in Figure 2.

Note that in Listing 5 the variables on the left-hand side of the equal sign should be understood as handles, which are used to indicate dependencies between actions. In particular, they do not represent output data generated by an action. The handles are passed as arguments in the second brackets to set

```
it    = Iterator(iterType='leaves')
dIn   = ConcurrentDataBegin()(it)

aR    = Action(routine='function_R')()
aS    = Action(routine='function_S')(aR)
aT    = Action(routine='function_T')(aS)

aX    = Action(routine='function_X')(aS)
aY    = Action(routine='function_Y')(aX)
aZ    = Action(routine='function_Z')([aT,aY])

dOut  = ConcurrentDataEnd()(aZ)
```

Listing 6: Example of Iterator and Concurrent Data patterns.

```
aX = Action(routine='function_X')()
aC = ConcurrentActions([Action(routine='function_R'),
                        Action(routine='function_S'),
                        Action(routine='function_T')])(aX)
aY = Action(routine='function_Y')(aC)
```

Listing 7: Example of Concurrent Actions pattern.

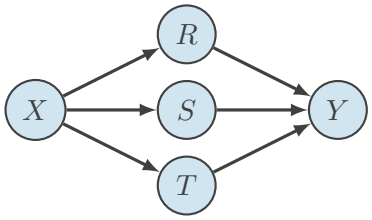

Figure 3: Example control flow graph of Concurrent Actions pattern, corresponding to Listing 7.

a dependency. The arguments passed with the keyword "routine" are function names of implemented functions. We denote the functions that implement an action as *action routines*.

Pattern: Iterator An Iterator defines which blocks of the computational mesh are passed to actions and, more generally, to pipelines of actions. Typically Flash-X iterators iterate over blocks of the mesh that are leaves or levels of the adaptively refined mesh.

Note that setting up an Iterator corresponds to setting up the "Action Parallel Distributor" of the runtime [10].

Pattern: Concurrent Data The Concurrent Data pattern describes that a single action, or a pipeline of actions, is executed on independent data items. While the Iterator provides the data source to pipelines, the Concurrent Data is the link between an Iterator and a pipeline, therefore it depends on the Iterator. We extend the previous example in Listing 5.

Pattern: Concurrent Actions The Concurrent Actions pattern describes that independent actions are executed on a single data item. While Listing 5 demonstrated concurrent actions by setting dependency handles appropriately, we also provide this pattern for explicitly defining concurrent actions. It is illustrated in Listing 7.

```
it    = Iterator(iterType='leaves')
dIn   = ConcurrentDataBegin()(it)

aR    = Action(routine='function_R')()
aS    = Action(routine='function_S')(aR)
aT    = Action(routine='function_T')(aS)

aX    = Action(routine='function_X')(aS)
aY    = Action(routine='function_Y')(aX)
aZ    = Action(routine='function_Z')([aT,aY])

dOut  = ConcurrentDataEnd()(aZ)

ConcurrentHardware(CPU={'actions': [aR,aS,aT,aZ]},
                   GPU={'actions': [aX,aY]})
```

Listing 8: Example of Concurrent Hardware pattern.

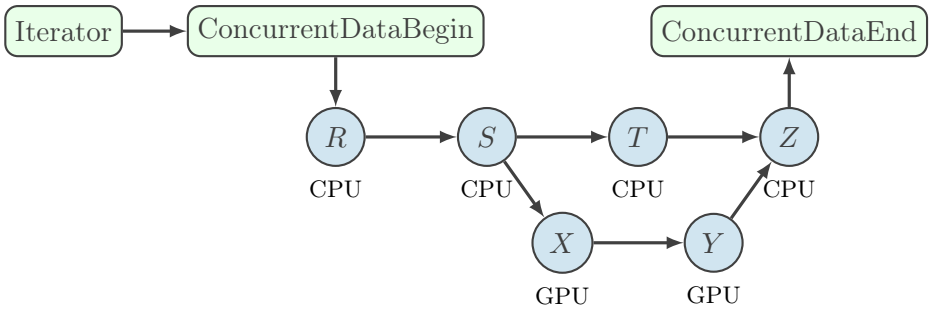

Figure 4: Example control flow graph of Concurrent Hardware pattern, corresponding to Listing 8, where node attributes define the device that an action is executed on, CPU or GPU here.

Pattern: Concurrent Hardware The Concurrent Hardware pattern is designed to establish a unique mapping between actions of a pipelines and heterogeneous hardware of a compute node. We express the mapping of actions to the hardware/device, on which an action will be executed, in an orthogonal manner by a separate syntax. Here, the handles of actions are utilized again.

The Concurrent Hardware pattern is demonstrated in Listing 8, where we extend the previous Listing 6. In this example, we assume the hardware node to contain two compute devices, CPU and GPU, which are used as keyword arguments of `ConcurrentHardware()`.

Pattern: Concurrent Actions, Data, Hardware As the name of the pattern suggests, we are combining our previous patterns in order to realize the splitting of concurrent data items such that a certain amount of data items is dispatched to one device and the remaining items to another device (i.e., simultaneously Concurrent Data and Hardware). Additionally, we allow for the possibility of executing different actions on different devices (i.e., simultaneously Concurrent Actions and Data and Hardware).

We implement mechanisms to let users control the splitting ratio and we allow for the ratio to be undetermined at the code generation phase. In the latter case, the ratio will be determined by load balancing techniques of the runtime.

3.2 Recipes

Recipes are written by users of Flash-X in a DSL that realizes the patterns from Section 3.1. The motivation for developing the recipe DSL is to, on the one hand, enable users to express high-level operations of a particular Flash-X multi-physics simulation in a concise manner. On the other hand, the recipe DSL enables the development of new tools for performance portability.

The recipe DSL of CodeFlow for Flash-X is designed based on the following requirements:

(i) a recipe expresses one full time step of a Flash-X simulation,

(ii) a recipe consists of operations (e.g., actions) defined by the patterns of Section 3.1,

(iii) a recipe can contain operations involving code that is not executed by the runtime (e.g., calls to external libraries),

(iv) recipes are associated with a collection of source code templates (e.g., based on PSTs),

(v) transforming a recipe through a toolchain to source code needs to be transparent and allow for user interventions at intermediate steps.

To generate functional source code from recipes, actions need to be implemented for a target platform. To generate platform-specific code for this purpose, we have been developing AST-based tools, but they are not discussed in the present paper. An example recipe is provided by Listing 9 in Section 4.

3.3 Control flow graphs

Given a recipe from Section 3.2, CodeFlow generates the corresponding control flow graph that has the structure of a directed acyclic graph. The nodes of the graph represent operations and actions, and the graph's edges represent an order of operations, possibly due to dependencies. The edges are additionally used to store information about data movement, which is inferred from the nodes.

We define requirements that constitute a valid control flow graph for our purposes:

(i) the graph is a directed acyclic graph,

(ii) it has a unique root node R and a unique end node E,

(iii) the longest path of the graph starts at R and ends at E,

(iv) for any node U of the graph, there exists a path from R to E that visits U,

(v) nodes corresponding to actions have a hardware device assigned to them.

As a consequence of the requirements, we can assume that control flow graphs can be referred to by tuples of root and end nodes. The assignment of devices to actions is realized by storing device information as an attribute of a node. These node attributes can then be used to identify—for all edges—whether a device change occurs between adjacent nodes. For this, we perform one traversal of the graph. During this traversal, we also propagate data movement information through the graph and include this information to the attributes of edges.

Processing a given recipe to generated code that interfaces with the orchestration runtime [10] is outlined in the following three steps. First, a control flow graph is generated from a recipe, where each operation of the recipe corresponds to one node of the graph, and the nodes are assigned hardware/devices for their execution. Second, this graph is transformed into a hierarchical control flow graph of two levels, where the coarse level only contains edges of the original graph that are marked for

a change of devices. Third, the coarse level graph is parsed to source code, which represents the driver of a Flash-X simulation utilizing the runtime; this implies the choice of thread-team configuration pertaining to the runtime (see [10]). The finer levels consist of subgraphs of the original graph. They are used to parse device-specific source code that is executed by the runtime, hence, these represent a "patch code" between the static Flash-X routines and the runtime.

CodeFlow builds upon the NetworkX library [8] for our implementation of directed acyclic graphs.

Creating and setting up control flow graphs The recipe DSL implements a functional syntax where function calls with two pairs of brackets are utilized (e.g., `Action(...)(...)` in Listing 5). Internally, this is realized by a function returning another function that accepts dependencies as arguments. The dependencies are provided in the form of handles the second pair of brackets, where the handles internally refer to nodes of the control flow graph. Different recipe DSL commands insert different types of nodes into the graph, which distinguish themselves by having different objects associated to them, such as, Iterators, Actions, etc. In summary, each command of the DSL inserts a new node into the control flow graph and inserts edges pointing from dependent nodes to the new node.

Device assignments stemming from `ConcurrentHardware()` calls are realized by adding node attributes to nodes that are associated with actions. After the assignment of devices is completed, the control flow graph is ready for further processing and analysis by the algorithms described next.

After creating the graph and assigning devices based on a recipe, we proceed to the setup that consists of determining device changes between neighboring nodes of the control flow graph. This is carried out with one recursive traversal of the graph, which is described in Algorithm 3.1. During this traversal, we also gather data movement information and annotate the associated edges with this information. Next, the control flow graph is transformed into a hierarchical graph as detailed in the next paragraph.

Algorithm 3.1 Set up edges of control flow graph for device change

Function: setUp(node U, data movement information M)

```
initialize U to graph's root node, if U is not defined
initialize M to be empty, if M is not defined
if U has data usage information then
    update data movement M with data usage information from node U
end if
for all neighbors N of U do
    annotate edge (U → N) whether device is changed
    if device change between N and U then
        annotate edge (U → N) with data movement information M
    end if
    call setUp(N, M)
end for
```

Transforming to hierarchical control flow graphs The control flow graph is now transformed into a hierarchical graph of two levels, where the coarse level only contains edges of the original "flat" graph that are marked for a change of devices. Algorithm 3.2 shows the steps required for transformation to a hierarchical graph, where each of the steps invoke their own algorithm.

In the first step, the device change attribute that was assigned to all edges in Algorithm 3.1 serves to separate coarse edges E_c from edges E_s that are designated for subgraphs. The separation of

edges is detailed in Algorithm 3.3. The set of edges E_s is used to extract subgraphs as described in Algorithm 3.4, resulting in the set of graphs, G_s. In the final step of the transformation process to a hierarchical graph, Algorithm 3.5 assembles the two-level hierarchy of coarse graph and subgraphs G_s, where the connectivity of the coarse edges E_c is used to link nodes at the coarse level.

Algorithm 3.2 Create hierarchical control flow graph

Function: toHierarchicalGraph()

1: separate edges into coarse edges E_c with device change and others E_s ▷ see Algorithm 3.3
2: create subgraphs G_s from edges E_s without device change ▷ see Algorithm 3.4
3: assemble hierarchical graph from subgraphs G_s and coarse edges E_c ▷ see Algorithm 3.5

Algorithm 3.3 Separate edges into coarse edges E_c with device change and remaining edges E_s

Function: separateEdges()

```
1: initialize E_c = {} and E_s = {}
2: for all nodes U do
3:     δ ← False, initialize no device change
4:     for all neighbors N of U do
5:         if edge (U → N) has device change then
6:             δ ← True
7:             exit loop
8:         end if
9:     end for
10:    if δ then
11:        for all neighbors N of U do
12:            add edge (U → N) to E_c
13:        end for
14:    else
15:        for all neighbors N of U do
16:            add edge (U → N) to E_s
17:        end for
18:    end if
19: end for
20: return E_c, E_s
```

Parsing source code from hierarchical control flow graphs After transforming a "flat" control flow graph into a hierarchical graph, we are ready for the final stage of the code generation pipeline, namely, to extract source code from the graph. The parsing of source code is performed by traversing the hierarchical control flow graph and calling a code assembler at each coarse node. This generates source code for the driver of Flash-X including code governing the invocation of the runtime. During the traversal of the coarse levels, each subgraph can generate code that implements runtime actions. Each edge of the coarse graph that contains data movement information triggers source code generation for data packets.

With these three components of source code generation for driver code, action related code, and data packet related code, we satisfy the interface required for using the runtime. Therefore, the source code for a Flash-X simulation is complete and we can proceed to the compilation phase.

Algorithm 3.4 Create subgraphs using edges E_s without device change

Function: createSubGraphs(edges E_s)

create subgraph H from edges E_s without device change, and their adjacent nodes
create multiple subgraphs G_s by splitting H into weakly connected components
create single-node subgraphs and add to G_s
return G_s

Algorithm 3.5 Assemble hierarchical graph from subgraphs G_s and coarse edges E_c

Function: assembleHierarchy(subgraphs G_s, edges E_c)

initialize empty graph H
for all subgraphs $G \in G_s$ **do**
    add new (coarse) node U_c to H that contains graph G
end for
for all edges $E = (S \to T) \in E_c$ **do**
    find node $S_c \in H$ such that S is an end node of a subgraph contained in coarse node S_c
    find node $T_c \in H$ such that T is a root node of a subgraph contained in coarse node T_c
    add edge $(S_c \to T_c)$ to H
end for
return H

4 Sedov example demonstrating a proof of concept

The Sedov problem is a hydrodynamic simulation with strong shocks and a nonplanar symmetry. The problem consists of evolving a cylindrical/spherical blast wave from a pointwise perturbation of the pressure in a homogeneous medium. For our proof of concept, we use Flash-X's simple unsplit method and HLL-type Riemann solver to solve the Sedov problem.

We are given static routines as a GPU implementation and a CPU implementation. The goal is to generate source code for the driver that utilizes the runtime with a target thread-team configuration, which consists of an action pipeline utilizing a GPU for a sequence of actions and a CPU for other actions. Utilizing the runtime entails the generation of source code for subroutines whose execution is managed by the runtime.

To setup CodeFlow for this code generation task, we create PST templates for the driver code and use a collection of templates for the subroutine code in order to encode platform-dependent source code at multiple different levels. Due to the flexible composability of PSTs, only a handful of templates (five in this example) are required to realize various platform-specific action pipelines.

Now, we use the recipe in Listing 9 to describe the actions of the simulation, their dependencies, and the mapping of actions to hardware devices. This recipe is processed by CodeFlow into a control flow graph, shown at the top of Figure 5. Node 0 represents the graph's root, and nodes 1 and 2 are the Iterator and ConcurrentDataBegin in the recipe, respectively. They are followed by four actions that are mapped to the GPU and one action designated to the CPU. The final node refers to ConcurrentDataEnd in the recipe. The "flat" control flow graph is transformed into a hierarchical graph, of which we show the coarse nodes in the middle of Figure 5, where all GPU actions are gathered into a subgraph. The coarse level of the hierarchical control flow graph is matched to the graph corresponding to our target thread-team configuration (bottom graph in Figure 5). The thread-team configuration is illustrated from the perspective of the runtime in Figure 6.

After the processing of graphs, CodeFlow can assemble the full PSTs of the driver, the GPU subroutine, and the CPU subroutine. These three PSTs are provided in Appendix A in Listings 10

to 12. Finally, the PSTs are parsed into code, which is given in Listings 13 to 15. Note that the data packet related code is not generated in this proof-of-concept example, because we use a generic data packet implementation. The generation of data packet source code that is tailored to the simulation is ongoing research.

5 Discussion and future work

The CodeFlow toolchain for Flash-X successfully realizes recipe-to-source code transformation of an example Flash-X simulation, the Sedov problem. In the future, we are extending the feature set of CodeFlow to be able to accommodate all use cases of multiphysics Flash-X simulations, and we are refining the design of CodeFlow to adapt it to the needs of users. More specifically, we are targeting the following future work for individual tools of CodeFlow.

Future work on parametrized source trees Our goal is to develop verification and debugging functionalities for PSTs. The correctness of a tree can be verified by checking its structure and by asserting the existence of all parameter definitions. We plan to augment parsed code from a tree with debugging information and precise references to the templates that are responsible for each line of code. Overall, our goal is to create a transparent way for tracing errors in generated code, such that users remain in control of each step of the code transformation pipeline.

Future work on recipe DSL We plan to simplify recipe writing by removing the requirement to state the arguments of action routines (`args=[...]` in Listing 9). This can be achieved by analyzing ASTs of the definitions of the routines.

We are discussing with the community what requirements the DSL for Flash-X recipes needs to satisfy. We are also experimenting with a custom syntax that is not borrowed from Python and that can, therefore, be tailored to Flash-X.

Future work on control flow graphs When building the control flow graph, we plan to extend the graph by hierarchically including subgraphs of lower-level recipes, utilizing Flash-X's hierarchical code configuration system.

We want to develop debugging functionality for tools that handle control flow graphs and, we want to increase transparency for users by generating plots of graphs.

Our goal is to gather all information about input and output data of subroutines in a dictionary and to use this dictionary to analyze the data movement in the the control flow graph. Furthermore, we plan to generate optimized source code for data packing and movement.

References

[1] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. IEEE, 2019.

[2] M. Bysiek, A. Drozd, and S. Matsuoka. Migrating legacy Fortran to Python while retaining Fortran-level performance through transpilation and type hints. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 9–18, 2016.

```
it    = Iterator(iterType='leaves')
dIn   = ConcurrentDataBegin(Uin=['DENS', 'VELX', 'VELY', 'VELZ', 'PRES', 'ENER'],
                            scratch=['flX', 'flY', 'flZ', 'auxC'])(it)
hySp  = Action(routine='hy_computeSoundSpeedHll',
               args=['lo', 'hi', 'U', 'auxC'])(dIn)
hyFl  = Action(routine='hy_computeFluxesHll',
               args=['dt', 'lo', 'hi', 'deltas', 'U', 'flX', 'flY', 'flZ', 'auxC'])(hySp)
hyUp  = Action(routine='hy_updateSolutionHll',
               args=['lo', 'hi', 'U', 'flX', 'flY', 'flZ'])(hyFl)
eosG  = Action(routine='eos_idealGammaDensIe',
               args=['lo', 'hi', 'U'])(hyUp)
ioIq  = Action(routine='io_computeIntegralQuantitiesByBlock',
               args=['tId', 'lo', 'hi', 'volumes', 'U'])(eosG)
dOut  = ConcurrentDataEnd(Uout=['DENS', 'VELX', 'VELY', 'VELZ', 'PRES', 'ENER'])(ioIq)

ConcurrentHardware(
    CPU={'nInitialThreads': 1, 'nTilesPerPacket':  0, 'actions': [ioIq]},
    GPU={'nInitialThreads': 5, 'nTilesPerPacket': 80, 'actions': [hySp, hyFl, hyUp, eosG]}
)
```

Listing 9: Recipe for Sedov example.

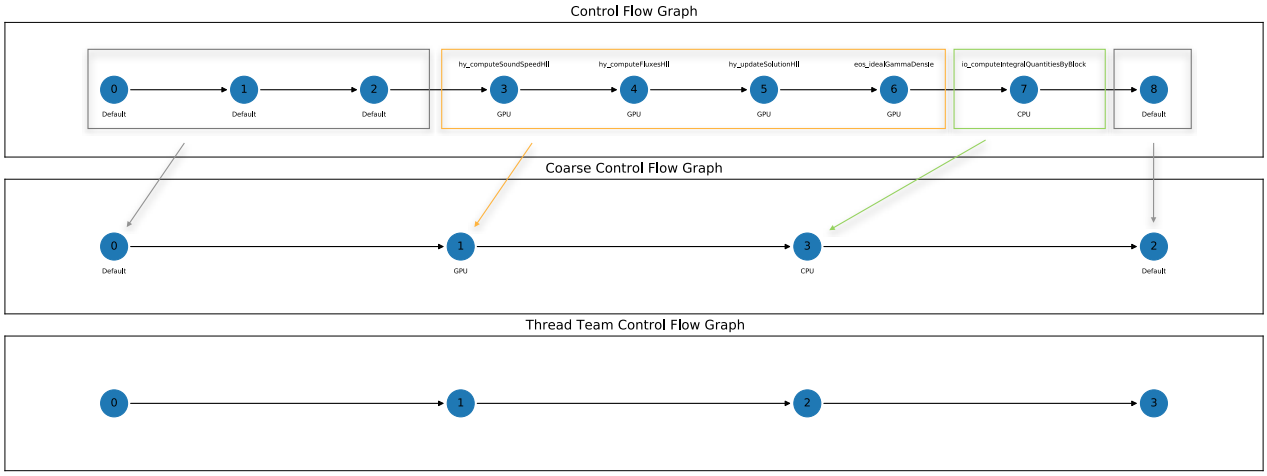

Figure 5: Control flow graphs of Sedov example. Top: graph generated from recipe; middle: coarse nodes of hierarchical control flow graph; bottom: reference graph of the thread-team configuration that was matched to coarse graph (middle).

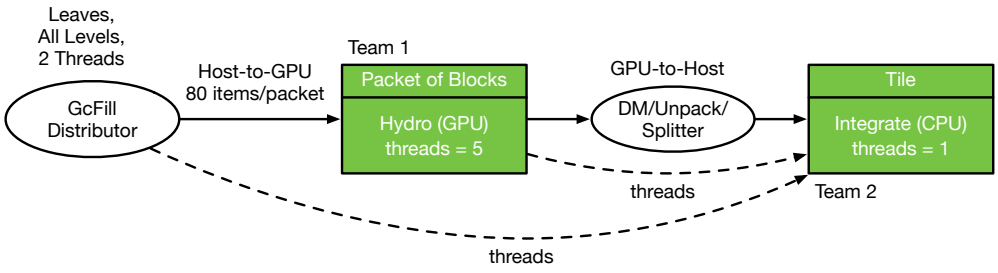

Figure 6: Thread-team configuration corresponding to the Sedov example, now illustrated from the perspective of the runtime and with additional runtime parameters. Multi-threaded CPUs allow for thread inheritance (depicted by dashed arrows) once action bundles (green boxes) have been completed.

[3] M. Bysiek, M. Wahib, A. Drozd, and S. Matsuoka. Towards portable high performance in Python: Transpilation, high-level IR, code transformations and compiler directives. Technical Report 38, Tokyo Institute of Technology, National Institute of Advanced Industrial Science and Technology, RIKEN Center for Computational Science, July 2018.

[4] P. H. Dave and H. B. Dave. *Compilers: principles and practice*. Pearson Education India, 2012.

[5] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10):512–522, 2009.

[6] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[7] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[8] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008.

[9] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[10] J. O'Neal, M. Wahib, A. Dubey, K. Weide, and T. Klosterman. Domain-specific runtime to orchestrate computation on heterogeneous platforms. Technical Report ANL-20/77, Argonne National Laboratory, Lemont, IL, 2021.

[11] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, et al. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370–1370, 2019.

A Generated PSTs and source code for Sedov example

```
{
  "_code": [
    "#include <cstdio>",
    "#include <string>",
    "",
    "#include <mpi.h>",
    "",
    "#include \"Io.h\"",
    "#include \"Hydro.h\"",
    "#include \"Driver.h\"",
    "#include \"Simulation.h\"",
    "",
    "#include \"Grid_REAL.h\"",
    "#include \"Grid.h\"",
    "#include \"Runtime.h\"",
    "#include \"OrchestrationLogger.h\"",
    "",
    "#include \"errorEstBlank.h\"",
    "",
    "#include \"Flash_par.h\"",
    "",
    "int main(int argc, char* argv[]) {",
    "    // TODO: Add in error handling code",
    "",
    "    //----- MIMIC Driver_init",
    "    // Analogous to calling Log_init",
    "    orchestration::Logger::instantiate(rp_Simulation::LOG_FILENAME);",
    "",
    "    // Analogous to calling Orchestration_init",
    "    orchestration::Runtime::instantiate(rp_Runtime::N_THREAD_TEAMS,",
    "                                        rp_Runtime::N_THREADS_PER_TEAM,",
    "                                        rp_Runtime::N_STREAMS,",
    "                                        rp_Runtime::MEMORY_POOL_SIZE_BYTES);",
    "",
    "    // Analogous to calling Grid_init",
    "    orchestration::Grid::instantiate();",
    "",
    "    // Analogous to calling IO_init",
    "    orchestration::Io::instantiate(rp_Simulation::INTEGRAL_QUANTITIES_FILENAME);",
    "",
    "    int  rank = 0;",
    "    MPI_Comm_rank(GLOBAL_COMM, &rank);",
    "",
    "    //----- MIMIC Grid_initDomain",
    "    orchestration::Io&       io      = orchestration::Io::instance();",
    "    orchestration::Grid&     grid    = orchestration::Grid::instance();",
    "    orchestration::Logger&   logger  = orchestration::Logger::instance();",
    "    orchestration::Runtime&  runtime = orchestration::Runtime::instance();",
    "    /* _param:runtime = runtime */",
    "    /* _param:a = A, _param:b B; _param:c = C */",
    "",
    "    Driver::dt      = rp_Simulation::DT_INIT;",
    "    Driver::simTime = rp_Simulation::T_0;",
    "",
    "    logger.log(\"[Simulation] Generate mesh and set initial conditions\");",
    "    grid.initDomain(Simulation::setInitialConditions_tile_cpu,",
    "                    rp_Simulation::N_THREADS_FOR_IC,",
    "                    Simulation::errorEstBlank);",
    "",
    "    //----- OUTPUT RESULTS TO FILES",
    "    // This only makes sense if the iteration is over LEAF blocks.",
    "    RuntimeAction     computeIntQuantitiesByBlk;",
    "",
    "    // TODO: Shouldn't this be done through the IO unit?",
    "    grid.writePlotfile(rp_Simulation::NAME + \"_plt_ICs\");",
    "",
    "    // Compute local integral quantities",
    "    runtime.executeCpuTasks(\"IntegralQ\", computeIntQuantitiesByBlk);",
    "    // Compute global integral quantities via DATA MOVEMENT",
    "    io.reduceToGlobalIntegralQuantities();",
    "    io.writeIntegralQuantities(Driver::simTime);",
    "",
    "    //----- SETUP ACTIONS",
    "    {
      "_param:indent": 1,
      "_link:setup": [
        {
          "_code": [
            "RuntimeAction  action_id1_GPU;",
            "action_id1_GPU.name            = \"id1_GPU\";",
            "action_id1_GPU.nInitialThreads = 16;",
            "action_id1_GPU.teamType        = ThreadTeamDataType::SET_OF_BLOCKS;",
            "action_id1_GPU.nTilesPerPacket = 64;",
            "action_id1_GPU.routine         = subroutine_id1_GPU;"
          ]
        },
        {
          "_code": [
            "RuntimeAction  action_id3_CPU;",
```

```
            "action_id3_CPU.name            = \"id3_CPU\";",
            "action_id3_CPU.nInitialThreads = 4;",
            "action_id3_CPU.teamType        = ThreadTeamDataType::BLOCK;",
            "action_id3_CPU.nTilesPerPacket = 0;",
            "action_id3_CPU.routine         = subroutine_id3_CPU;"
          ]
        }
      ]
    },
    "",
    "    //----- MIMIC Driver_evolveFlash",
    "    logger.log(\"[Simulation] \" + rp_Simulation::NAME + \" simulation started\");",
    "",
    "    unsigned int   nStep   = 1;",
    "    while ((nStep <= rp_Simulation::MAX_STEPS) && (Driver::simTime < rp_Simulation::T_MAX)) {",
    "",
    "        // TODO: Log as well",
    "        if (rank == MASTER_PE) {",
    "            printf(\"Step n=%d / t=%.4e / dt=%.4e\\n\", nStep, Driver::simTime, Driver::dt);",
    "        }",
    "",
    "        //----- ADVANCE SOLUTION BASED ON HYDRODYNAMICS",
    "        if (nStep > 1) {",
    "            grid.fillGuardCells();",
    "        }",
    "",
    "        //----- EXECUTE ACTIONS, single block",
    {
      "_param:indent": 2,
      "_link:execute": [
        {
          "_code": [
            "_param:runtime.executeExtendedGpuTasks(\"Action Pipeline\", action_id1_GPU, action_id3_CPU);"
          ]
        }
      ]
    },
    "",
    "        //----- EXECUTE ACTIONS, interleaved with bookkeeping",
    "        /* _HIDE_link:execute_recipe_1 */",
    "        //[",
    "        //  {'_HIDE_code': hydro}",
    "        //  {'_HIDE_code': hypre}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //]",
    "        /* driver bookkeeping */",
    "        /* _HIDE_link:execute_recipe_2 */",
    "        //[",
    "        //  {'_HIDE_code': hydro}",
    "        //  {'_HIDE_code': hypre}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //  {'_HIDE_code': ...}",
    "        //]",
    "",
    "        //----- OUTPUT RESULTS TO FILES",
    "        io.reduceToGlobalIntegralQuantities();",
    "        io.writeIntegralQuantities(Driver::simTime);",
    "",
    "        if (!(nStep % rp_Driver::WRITE_EVERY_N_STEPS)) {",
    "            grid.writePlotfile(rp_Simulation::NAME + \"_plt_\" + std::to_string(nStep));",
    "        }",
    "",
    "        //----- UPDATE GRID IF REQUIRED",
    "        // We are running in pseudo-UG for now and can therefore skip this",
    "",
    "        //----- COMPUTE dt FOR NEXT STEP",
    "        // NOTE: The AllReduce that follows should appear here",
    "        //       rather than be buried in Driver_computeDt.",
    "        //",
    "        // When this problem is run in FLASH-X, the hydro dt is always greater",
    "        // than 5.0e-5 seconds.  Therefore, using a dt value fixed to a smaller",
    "        // value should always keep us on the stable side of the CFL condition.",
    "        // Therefore, we skip the computeDt for Hydro here.",
    "        //",
    "        // When a dt value of 5.0e-5 is used, FLASH-X complains that it is too",
    "        // low and sets dt to the Hydro CFL-determined dt value, which should be",
    "        // Simulation::DT_INIT.  There after, it allows for 5.0e-5.  Therefore,",
    "        // we mimic that dt sequence here so that we can directly compare",
    "        // results.",
    "        Driver::dt = rp_Driver::DT_AFTER;",
    "",
    "        ++nStep;",
    "",
    "    }",
    "    logger.log(\"[Simulation] \" + rp_Simulation::NAME + \" simulation terminated\");",
    "    if (Driver::simTime >= rp_Simulation::T_MAX) {",
    "        Logger::instance().log(\"[Simulation] Reached max SimTime\");",
    "    }",
```

```
    "    grid.writePlotfile(rp_Simulation::NAME + \"_plt_final\");",
    "",
    "    nStep = std::min(nStep, rp_Simulation::MAX_STEPS);",
    "",
    "    //----- CLEAN-UP",
    "    // The singletons are finalized automatically when the program is",
    "    // terminating.",
    "",
    "    return 0;",
    "}"
  ],
  "_param:runtime": "runtime",
  "_param:a": "A",
  "_param:b": "B",
  "_param:c": "C",
  "_param:__file__": "ex_sedov_driver_Default.cpp"
}
```

Listing 10: Sedov example: Generated PST of driver.

```
{
  "_code": [
    "#include \"Eos.h\"",
    "#include \"Hydro.h\"",
    "#include \"Driver.h\"",
    "#include \"Flash.h\"",
    {
      "_link:setup": [
        {
          "_code": [
            "#ifndef ENABLE_OPENACC_OFFLOAD",
            "#error \"This file should only be compiled if using OpenACC offloading\"",
            "#endif",
            "#include \"DataPacket.h\"",
            "#include \"StreamManager.h\""
          ],
          "_param:__file__": "ex_sedov_subroutine_GPU_main.json"
        }
      ]
    },
    "void _param:functionName(const int _param:tId, orchestration::DataItem* _param:dataItem)",
    "{",
    "    using namespace orchestration;",
    {
      "_param:indent": 1,
      "_link:execute": [
        {
          "_param:queue": "queue_h",
          "_param:nTiles": "nTiles_d",
          "_param:contents": "contents_d",
          "_param:dt": "dt_d",
          "_code": [
            "DataPacket*                packet_h = dynamic_cast<DataPacket*>(_param:dataItem);",
            "const PacketDataLocation   location = packet_h->getDataLocation();",
            "const int                  _param:queue    = packet_h->asynchronousQueue();",
            "const std::size_t*         _param:nTiles   = packet_h->nTilesGpu();",
            "const PacketContents*      _param:contents = packet_h->tilePointers();",
            "const Real*                _param:dt       = packet_h->timeStepGpu();",
            "packet_h->setVariableMask(UNK_VARS_BEGIN_C, UNK_VARS_END_C);",
            {
              "_link:setup": []
            },
            "#pragma acc data deviceptr(_param:nTiles, _param:contents, _param:dt)",
            "{",
            "    if (location == PacketDataLocation::CC1) {",
            {
              "_param:indent": 2,
              "_param:pointer_U": "CC1_d",
              "_param:pointer_auxC": "CC2_d",
              "_link:execute": [
                {
                  "_param:iterIndex": "n",
                  "_code": [
                    "#pragma acc parallel loop gang default(none) async(_param:queue)",
                    "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {"
,
                    {
                      "_param:indent": 1,
                      "_link:execute": [
                        {
                          "_param:U": "U_d",
                          "_param:auxC": "auxC_d",
                          "_param:flX": "ptrs->FCX_d",
                          "_param:flY": "ptrs->FCY_d",
                          "_param:flZ": "ptrs->FCZ_d",
                          "_param:lo": "ptrs->lo_d",
                          "_param:hi": "ptrs->hi_d",
                          "_param:deltas": "ptrs->deltas_d",
                          "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                          "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                          "_code": [
                            "const PacketContents*  ptrs = _param:contents + _param:iterIndex;",
                            {
                              "_link:setup": [
                                {
                                  "_code": [
                                    "FArray4D*  _param:U    = ptrs->_param:pointer_U;",
                                    "FArray4D*  _param:auxC = ptrs->_param:pointer_auxC;"
                                  ]
                                }
                              ]
                            },
                            {
                              "_link:execute": [
                                {
                                  "_code": [
                                    "hy_computeSoundSpeedHll_GPU(_param:lo, _param:hi, _param:U, _param:auxC
);"
                                  ]
                                }
                              ]
                            }
                          ],
```

```
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              },
              {
                "_param:iterIndex": "n",
                "_code": [
                  "#pragma acc parallel loop gang default(none) async(_param:queue)",
                  "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                  {
                    "_param:indent": 1,
                    "_link:execute": [
                      {
                        "_param:U": "U_d",
                        "_param:auxC": "auxC_d",
                        "_param:flX": "ptrs->FCX_d",
                        "_param:flY": "ptrs->FCY_d",
                        "_param:flZ": "ptrs->FCZ_d",
                        "_param:lo": "ptrs->lo_d",
                        "_param:hi": "ptrs->hi_d",
                        "_param:deltas": "ptrs->deltas_d",
                        "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                        "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                        "_code": [
                          "const PacketContents* ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                                  "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
                                "_code": [
                                  "hy_computeFluxesHll_GPU(_param:dt, _param:lo, _param:hi, _param:deltas, _param:U, _param:flX, _param:flY, _param:flZ, _param:auxC);"
                                ]
                              }
                            ]
                          }
                        ],
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              },
              {
                "_param:iterIndex": "n",
                "_code": [
                  "#pragma acc parallel loop gang default(none) async(_param:queue)",
                  "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                  {
                    "_param:indent": 1,
                    "_link:execute": [
                      {
                        "_param:U": "U_d",
                        "_param:auxC": "auxC_d",
                        "_param:flX": "ptrs->FCX_d",
                        "_param:flY": "ptrs->FCY_d",
                        "_param:flZ": "ptrs->FCZ_d",
                        "_param:lo": "ptrs->lo_d",
                        "_param:hi": "ptrs->hi_d",
                        "_param:deltas": "ptrs->deltas_d",
                        "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                        "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                        "_code": [
                          "const PacketContents* ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D* _param:U    = ptrs->_param:pointer_U;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
```

```
                                    "_code": [
                                      "hy_updateSolutionHll_GPU(_param:lo, _param:hi, _param:U, _param:flX, _param:flY, _param:flZ);"
                                    ]
                                  }
                                ]
                              }
                            ],
                            "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                          }
                        ]
                      },
                      "}"
                    ],
                    "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
                  },
                  {
                    "_param:iterIndex": "n",
                    "_code": [
                      "#pragma acc parallel loop gang default(none) async(_param:queue)",
                      "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                      {
                        "_param:indent": 1,
                        "_link:execute": [
                          {
                            "_param:U": "U_d",
                            "_param:auxC": "auxC_d",
                            "_param:flX": "ptrs->FCX_d",
                            "_param:flY": "ptrs->FCY_d",
                            "_param:flZ": "ptrs->FCZ_d",
                            "_param:lo": "ptrs->lo_d",
                            "_param:hi": "ptrs->hi_d",
                            "_param:deltas": "ptrs->deltas_d",
                            "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                            "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                            "_code": [
                              "const PacketContents* ptrs = _param:contents + _param:iterIndex;",
                              {
                                "_link:setup": [
                                  {
                                    "_code": [
                                      "FArray4D* _param:U    = ptrs->_param:pointer_U;"
                                    ]
                                  }
                                ]
                              },
                              {
                                "_link:execute": [
                                  {
                                    "_code": [
                                      "eos_idealGammaDensIe_GPU(_param:lo, _param:hi, _param:U);"
                                    ]
                                  }
                                ]
                              }
                            ],
                            "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                          }
                        ]
                      },
                      "}"
                    ],
                    "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
                  }
                ]
              },
              "    } else if (location == PacketDataLocation::CC2) {",
              {
                "_param:indent": 2,
                "_param:pointer_U": "CC2_d",
                "_param:pointer_auxC": "CC1_d",
                "_link:execute": [
                  {
                    "_param:iterIndex": "n",
                    "_code": [
                      "#pragma acc parallel loop gang default(none) async(_param:queue)",
                      "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                      {
                        "_param:indent": 1,
                        "_link:execute": [
                          {
                            "_param:U": "U_d",
                            "_param:auxC": "auxC_d",
                            "_param:flX": "ptrs->FCX_d",
                            "_param:flY": "ptrs->FCY_d",
                            "_param:flZ": "ptrs->FCZ_d",
                            "_param:lo": "ptrs->lo_d",
                            "_param:hi": "ptrs->hi_d",
                            "_param:deltas": "ptrs->deltas_d",
                            "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                            "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
```

```
                        "_code": [
                          "const PacketContents*  ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D*  _param:U    = ptrs->_param:pointer_U;",
                                  "FArray4D*  _param:auxC = ptrs->_param:pointer_auxC;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
                                "_code": [
                                  "hy_computeSoundSpeedHll_GPU(_param:lo, _param:hi, _param:U, _param:auxC);"
                                ]
                              }
                            ]
                          }
                        ],
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              },
              {
                "_param:iterIndex": "n",
                "_code": [
                  "#pragma acc parallel loop gang default(none) async(_param:queue)",
                  "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                  {
                    "_param:indent": 1,
                    "_link:execute": [
                      {
                        "_param:U": "U_d",
                        "_param:auxC": "auxC_d",
                        "_param:flX": "ptrs->FCX_d",
                        "_param:flY": "ptrs->FCY_d",
                        "_param:flZ": "ptrs->FCZ_d",
                        "_param:lo": "ptrs->lo_d",
                        "_param:hi": "ptrs->hi_d",
                        "_param:deltas": "ptrs->deltas_d",
                        "_param:setup_U": "FArray4D*  _param:U    = ptrs->_param:pointer_U;",
                        "_param:setup_auxC": "FArray4D*  _param:auxC = ptrs->_param:pointer_auxC;",
                        "_code": [
                          "const PacketContents*  ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D*  _param:U    = ptrs->_param:pointer_U;",
                                  "FArray4D*  _param:auxC = ptrs->_param:pointer_auxC;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
                                "_code": [
                                  "hy_computeFluxesHll_GPU(_param:dt, _param:lo, _param:hi, _param:deltas, _param:U, _param:flX, _param:flY, _param:flZ, _param:auxC);"
                                ]
                              }
                            ]
                          }
                        ],
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              },
              {
                "_param:iterIndex": "n",
                "_code": [
                  "#pragma acc parallel loop gang default(none) async(_param:queue)",
                  "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {",
                  {
                    "_param:indent": 1,
                    "_link:execute": [
                      {
                        "_param:U": "U_d",
```

```
                        "_param:auxC": "auxC_d",
                        "_param:flX": "ptrs->FCX_d",
                        "_param:flY": "ptrs->FCY_d",
                        "_param:flZ": "ptrs->FCZ_d",
                        "_param:lo": "ptrs->lo_d",
                        "_param:hi": "ptrs->hi_d",
                        "_param:deltas": "ptrs->deltas_d",
                        "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                        "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                        "_code": [
                          "const PacketContents* ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D* _param:U    = ptrs->_param:pointer_U;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
                                "_code": [
                                  "hy_updateSolutionHll_GPU(_param:lo, _param:hi, _param:U, _param:flX,
_param:flY, _param:flZ);"
                                ]
                              }
                            ]
                          }
                        ],
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              },
              {
                "_param:iterIndex": "n",
                "_code": [
                  "#pragma acc parallel loop gang default(none) async(_param:queue)",
                  "for (std::size_t _param:iterIndex=0; _param:iterIndex<*nTiles_d; ++_param:iterIndex) {"
,
                  {
                    "_param:indent": 1,
                    "_link:execute": [
                      {
                        "_param:U": "U_d",
                        "_param:auxC": "auxC_d",
                        "_param:flX": "ptrs->FCX_d",
                        "_param:flY": "ptrs->FCY_d",
                        "_param:flZ": "ptrs->FCZ_d",
                        "_param:lo": "ptrs->lo_d",
                        "_param:hi": "ptrs->hi_d",
                        "_param:deltas": "ptrs->deltas_d",
                        "_param:setup_U": "FArray4D* _param:U    = ptrs->_param:pointer_U;",
                        "_param:setup_auxC": "FArray4D* _param:auxC = ptrs->_param:pointer_auxC;",
                        "_code": [
                          "const PacketContents* ptrs = _param:contents + _param:iterIndex;",
                          {
                            "_link:setup": [
                              {
                                "_code": [
                                  "FArray4D* _param:U    = ptrs->_param:pointer_U;"
                                ]
                              }
                            ]
                          },
                          {
                            "_link:execute": [
                              {
                                "_code": [
                                  "eos_idealGammaDensIe_GPU(_param:lo, _param:hi, _param:U);"
                                ]
                              }
                            ]
                          }
                        ],
                        "_param:__file__": "ex_sedov_subroutine_GPU_action_kernel.json"
                      }
                    ]
                  },
                  "}"
                ],
                "_param:__file__": "ex_sedov_subroutine_GPU_action_loop.json"
              }
            ]
          },
          "    } else {",
          "      throw std::logic_error(\"[_param:functionName] Data location is not CC1 or CC2\");",
          "    }",
```

```
            "} // OpenACC data block",
            "#pragma acc wait(_param:queue)"
          ],
          "_param:__file__": "ex_sedov_subroutine_GPU_main.json"
        }
      ]
    },
    "}"
  ],
  "_param:functionName": "subroutine_id1_GPU",
  "_param:tId": "dataItem",
  "_param:dataItem": "dataItem",
  "_param:__file__": "ex_sedov_subroutine_Default_main.json"
}
```

Listing 11: Sedov example: Generated PST of GPU subroutine.

```
{
  "_code": [
    "#include \"Eos.h\"",
    "#include \"Hydro.h\"",
    "#include \"Driver.h\"",
    "#include \"Flash.h\"",
    {
      "_link:setup": [
        {
          "_code": [
            "#include \"Tile.h\""
          ],
          "_param:__file__": "ex_sedov_subroutine_CPU_main.json"
        }
      ]
    },
    "void _param:functionName(const int _param:tId, orchestration::DataItem* _param:dataItem)",
    "{",
    "    using namespace orchestration;",
    {
      "_param:indent": 1,
      "_link:execute": [
        {
          "_param:dt": "Driver::dt",
          "_param:lo": "tileDesc->lo()",
          "_param:hi": "tileDesc->hi()",
          "_param:deltas": "tileDesc->deltas()",
          "_param:level": "tileDesc->level()",
          "_param:U": "tileDesc->data()",
          "_code": [
            "Tile*  tileDesc = dynamic_cast<Tile*>(_param:dataItem);",
            {
              "_link:setup": []
            },
            {
              "_link:execute": [
                {
                  "_param:flX": "flX",
                  "_param:flY": "flX",
                  "_param:flZ": "flZ",
                  "_param:auxC": "auxC",
                  "_param:volumes": "volumes",
                  "_param:fHiX": "IntVect{LIST_NDIM(hi.I()+K1D, hi.J(), hi.K())}",
                  "_param:fHiY": "IntVect{LIST_NDIM(hi.I(), hi.J()+K2D, hi.K())}",
                  "_param:fHiZ": "IntVect{LIST_NDIM(hi.I(), hi.J(), hi.K()+K3D)}",
                  "_param:setup_flX": "FArray4D  _param:flX = FArray4D::buildScratchArray4D(lo, _param:fHiX, NFLUXES);",
                  "_param:setup_flY": "FArray4D  _param:flY = FArray4D::buildScratchArray4D(lo, _param:fHiY, NFLUXES);",
                  "_param:setup_flZ": "FArray4D  _param:flZ = FArray4D::buildScratchArray4D(lo, _param:fHiZ, NFLUXES);",
                  "_param:cLo": "IntVect{LIST_NDIM(lo.I()-K1D, lo.J()-K2D, lo.K()-K3D)}",
                  "_param:cHi": "IntVect{LIST_NDIM(hi.I()+K1D, hi.J()+K2D, hi.K()+K3D)}",
                  "_param:setup_auxC": "FArray3D  _param:auxC = FArray3D::buildScratchArray(_param:cLo, _param:cHi);",
                  "_param:setup_volumes": [
                    "Grid& grid = Grid::instance();",
                    "Real  volumes_buffer[  (_param:hi.I() - _param:lo.I() + 1)",
                    "                     * (_param:hi.J() - _param:lo.J() + 1)",
                    "                     * (_param:hi.K() - _param:lo.K() + 1)];",
                    "grid.fillCellVolumes(_param:level, _param:lo, _param:hi, volumes_buffer);",
                    "const FArray3D  _param:volumes = {volumes_buffer, _param:lo, _param:hi};"
                  ],
                  "_code": [
                    "{",
                    {
                      "_param:indent": 1,
                      "_link:setup": [
                        {
                          "_code": [
                            "Grid& grid = Grid::instance();",
                            "Real  volumes_buffer[  (_param:hi.I() - _param:lo.I() + 1)",
                            "                     * (_param:hi.J() - _param:lo.J() + 1)",
                            "                     * (_param:hi.K() - _param:lo.K() + 1)];",
                            "grid.fillCellVolumes(_param:level, _param:lo, _param:hi, volumes_buffer);",
                            "const FArray3D  _param:volumes = {volumes_buffer, _param:lo, _param:hi};"
                          ]
                        }
                      ],
                      "_link:execute": [
                        {
                          "_code": [
                            "io_computeIntegralQuantitiesByBlock_CPU(_param:tId, _param:lo, _param:hi, _param:volumes, _param:U);"
                          ]
                        }
                      ]
                    },
                    "}"
                  ],
                  "_param:__file__": "ex_sedov_subroutine_CPU_action_kernel.json"
                }
              ]
```

```
            }
          ],
          "_param:__file__": "ex_sedov_subroutine_CPU_main.json"
        }
      ]
    },
    "}"
  ],
  "_param:functionName": "subroutine_id3_CPU",
  "_param:tId": "dataItem",
  "_param:dataItem": "dataItem",
  "_param:__file__": "ex_sedov_subroutine_Default_main.json"
}
```

Listing 12: Sedov example: Generated PST of CPU subroutine.

```
/* <ex_sedov_driver_Default.cpp> */
#include <cstdio>
#include <string>

#include <mpi.h>

#include "Io.h"
#include "Hydro.h"
#include "Driver.h"
#include "Simulation.h"

#include "Grid_REAL.h"
#include "Grid.h"
#include "Runtime.h"
#include "OrchestrationLogger.h"

#include "errorEstBlank.h"

#include "Flash_par.h"

int main(int argc, char* argv[]) {
    // TODO: Add in error handling code

    //----- MIMIC Driver_init
    // Analogous to calling Log_init
    orchestration::Logger::instantiate(rp_Simulation::LOG_FILENAME);

    // Analogous to calling Orchestration_init
    orchestration::Runtime::instantiate(rp_Runtime::N_THREAD_TEAMS,
                                        rp_Runtime::N_THREADS_PER_TEAM,
                                        rp_Runtime::N_STREAMS,
                                        rp_Runtime::MEMORY_POOL_SIZE_BYTES);

    // Analogous to calling Grid_init
    orchestration::Grid::instantiate();

    // Analogous to calling IO_init
    orchestration::Io::instantiate(rp_Simulation::INTEGRAL_QUANTITIES_FILENAME);

    int  rank = 0;
    MPI_Comm_rank(GLOBAL_COMM, &rank);

    //----- MIMIC Grid_initDomain
    orchestration::Io&       io      = orchestration::Io::instance();
    orchestration::Grid&     grid    = orchestration::Grid::instance();
    orchestration::Logger&   logger  = orchestration::Logger::instance();
    orchestration::Runtime&  runtime = orchestration::Runtime::instance();
    /* runtime = runtime */
    /* A = A, B B; C = C */

    Driver::dt      = rp_Simulation::DT_INIT;
    Driver::simTime = rp_Simulation::T_0;

    logger.log("[Simulation] Generate mesh and set initial conditions");
    grid.initDomain(Simulation::setInitialConditions_tile_cpu,
                    rp_Simulation::N_THREADS_FOR_IC,
                    Simulation::errorEstBlank);

    //----- OUTPUT RESULTS TO FILES
    // This only makes sense if the iteration is over LEAF blocks.
    RuntimeAction     computeIntQuantitiesByBlk;

    // TODO: Shouldn't this be done through the IO unit?
    grid.writePlotfile(rp_Simulation::NAME + "_plt_ICs");

    // Compute local integral quantities
    runtime.executeCpuTasks("IntegralQ", computeIntQuantitiesByBlk);
    // Compute global integral quantities via DATA MOVEMENT
    io.reduceToGlobalIntegralQuantities();
    io.writeIntegralQuantities(Driver::simTime);

    //----- SETUP ACTIONS
    RuntimeAction  action_id1_GPU;
    action_id1_GPU.name           = "id1_GPU";
    action_id1_GPU.nInitialThreads = 16;
    action_id1_GPU.teamType        = ThreadTeamDataType::SET_OF_BLOCKS;
    action_id1_GPU.nTilesPerPacket = 64;
    action_id1_GPU.routine         = subroutine_id1_GPU;
    RuntimeAction  action_id3_CPU;
    action_id3_CPU.name           = "id3_CPU";
    action_id3_CPU.nInitialThreads = 4;
    action_id3_CPU.teamType        = ThreadTeamDataType::BLOCK;
    action_id3_CPU.nTilesPerPacket = 0;
    action_id3_CPU.routine         = subroutine_id3_CPU;

    //----- MIMIC Driver_evolveFlash
    logger.log("[Simulation] " + rp_Simulation::NAME + " simulation started");

    unsigned int    nStep   = 1;
    while ((nStep <= rp_Simulation::MAX_STEPS) && (Driver::simTime < rp_Simulation::T_MAX)) {

        // TODO: Log as well
        if (rank == MASTER_PE) {
```

```
94              printf("Step n=%d / t=%.4e / dt=%.4e\n", nStep, Driver::simTime, Driver::dt);
95          }
96
97          //------ ADVANCE SOLUTION BASED ON HYDRODYNAMICS
98          if (nStep > 1) {
99              grid.fillGuardCells();
100         }
101
102         //------ EXECUTE ACTIONS
103         runtime.executeExtendedGpuTasks("Action Pipeline", action_id1_GPU, action_id3_CPU);
104
105         //------ OUTPUT RESULTS TO FILES
106         io.reduceToGlobalIntegralQuantities();
107         io.writeIntegralQuantities(Driver::simTime);
108
109         if (!(nStep % rp_Driver::WRITE_EVERY_N_STEPS)) {
110             grid.writePlotfile(rp_Simulation::NAME + "_plt_" + std::to_string(nStep));
111         }
112
113         //------ UPDATE GRID IF REQUIRED
114         // We are running in pseudo-UG for now and can therefore skip this
115
116         //------ COMPUTE dt FOR NEXT STEP
117         // NOTE: The AllReduce that follows should appear here
118         //       rather than be buried in Driver_computeDt.
119         //
120         // When this problem is run in FLASH-X, the hydro dt is always greater
121         // than 5.0e-5 seconds.  Therefore, using a dt value fixed to a smaller
122         // value should always keep us on the stable side of the CFL condition.
123         // Therefore, we skip the computeDt for Hydro here.
124         //
125         // When a dt value of 5.0e-5 is used, FLASH-X complains that it is too
126         // low and sets dt to the Hydro CFL-determined dt value, which should be
127         // Simulation::DT_INIT.  There after, it allows for 5.0e-5.  Therefore,
128         // we mimic that dt sequence here so that we can directly compare
129         // results.
130         Driver::dt = rp_Driver::DT_AFTER;
131
132         ++nStep;
133
134     }
135     logger.log("[Simulation] " + rp_Simulation::NAME + " simulation terminated");
136     if (Driver::simTime >= rp_Simulation::T_MAX) {
137         Logger::instance().log("[Simulation] Reached max SimTime");
138     }
139     grid.writePlotfile(rp_Simulation::NAME + "_plt_final");
140
141     nStep = std::min(nStep, rp_Simulation::MAX_STEPS);
142
143     //------ CLEAN-UP
144     // The singletons are finalized automatically when the program is
145     // terminating.
146
147     return 0;
148 }
149 /* </ex_sedov_driver_Default.cpp> */
```

Listing 13: Sedov example: Generated C++ code of driver.

```
/* <ex_sedov_subroutine_Default_main.json> */
#include "Eos.h"
#include "Hydro.h"
#include "Driver.h"
#include "Flash.h"
/* <ex_sedov_subroutine_GPU_main.json> */
#ifndef ENABLE_OPENACC_OFFLOAD
#error "This file should only be compiled if using OpenACC offloading"
#endif
#include "DataPacket.h"
#include "StreamManager.h"
/* </ex_sedov_subroutine_GPU_main.json> */
void subroutine_id1_GPU(const int dataItem, orchestration::DataItem* dataItem)
{
    using namespace orchestration;
    /* <ex_sedov_subroutine_GPU_main.json> */
    DataPacket*               packet_h = dynamic_cast<DataPacket*>(dataItem);
    const PacketDataLocation  location = packet_h->getDataLocation();
    const int                 queue_h  = packet_h->asynchronousQueue();
    const std::size_t*        nTiles_d = packet_h->nTilesGpu();
    const PacketContents*     contents_d = packet_h->tilePointers();
    const Real*               dt_d     = packet_h->timeStepGpu();
    packet_h->setVariableMask(UNK_VARS_BEGIN_C, UNK_VARS_END_C);
    #pragma acc data deviceptr(nTiles_d, contents_d, dt_d)
    {
        if (location == PacketDataLocation::CC1) {
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC1_d;
                FArray4D*  auxC_d = ptrs->CC2_d;
                hy_computeSoundSpeedHll_GPU(ptrs->lo_d, ptrs->hi_d, U_d, auxC_d);
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
            /* </ex_sedov_subroutine_GPU_action_loop.json> */
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC1_d;
                FArray4D*  auxC_d = ptrs->CC2_d;
                hy_computeFluxesHll_GPU(dt_d, ptrs->lo_d, ptrs->hi_d, ptrs->deltas_d, U_d, ptrs->FCX_d, ptrs
    ->FCY_d, ptrs->FCZ_d, auxC_d);
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
            /* </ex_sedov_subroutine_GPU_action_loop.json> */
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC1_d;
                hy_updateSolutionHll_GPU(ptrs->lo_d, ptrs->hi_d, U_d, ptrs->FCX_d, ptrs->FCY_d, ptrs->FCZ_d)
    ;
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
            /* </ex_sedov_subroutine_GPU_action_loop.json> */
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC1_d;
                eos_idealGammaDensIe_GPU(ptrs->lo_d, ptrs->hi_d, U_d);
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
            /* </ex_sedov_subroutine_GPU_action_loop.json> */
        } else if (location == PacketDataLocation::CC2) {
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC2_d;
                FArray4D*  auxC_d = ptrs->CC1_d;
                hy_computeSoundSpeedHll_GPU(ptrs->lo_d, ptrs->hi_d, U_d, auxC_d);
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
            /* </ex_sedov_subroutine_GPU_action_loop.json> */
            /* <ex_sedov_subroutine_GPU_action_loop.json> */
            #pragma acc parallel loop gang default(none) async(queue_h)
            for (std::size_t n=0; n<*nTiles_d; ++n) {
                /* <ex_sedov_subroutine_GPU_action_kernel.json> */
                const PacketContents*  ptrs = contents_d + n;
                FArray4D*  U_d    = ptrs->CC2_d;
                FArray4D*  auxC_d = ptrs->CC1_d;
                hy_computeFluxesHll_GPU(dt_d, ptrs->lo_d, ptrs->hi_d, ptrs->deltas_d, U_d, ptrs->FCX_d, ptrs
    ->FCY_d, ptrs->FCZ_d, auxC_d);
                /* </ex_sedov_subroutine_GPU_action_kernel.json> */
            }
```

```
91              /* </ex_sedov_subroutine_GPU_action_loop.json> */
92              /* <ex_sedov_subroutine_GPU_action_loop.json> */
93              #pragma acc parallel loop gang default(none) async(queue_h)
94              for (std::size_t n=0; n<*nTiles_d; ++n) {
95                  /* <ex_sedov_subroutine_GPU_action_kernel.json> */
96                  const PacketContents*  ptrs = contents_d + n;
97                  FArray4D*  U_d    = ptrs->CC2_d;
98                  hy_updateSolutionHll_GPU(ptrs->lo_d, ptrs->hi_d, U_d, ptrs->FCX_d, ptrs->FCY_d, ptrs->FCZ_d)
    ;
99                  /* </ex_sedov_subroutine_GPU_action_kernel.json> */
100             }
101             /* </ex_sedov_subroutine_GPU_action_loop.json> */
102             /* <ex_sedov_subroutine_GPU_action_loop.json> */
103             #pragma acc parallel loop gang default(none) async(queue_h)
104             for (std::size_t n=0; n<*nTiles_d; ++n) {
105                 /* <ex_sedov_subroutine_GPU_action_kernel.json> */
106                 const PacketContents*  ptrs = contents_d + n;
107                 FArray4D*  U_d    = ptrs->CC2_d;
108                 eos_idealGammaDensIe_GPU(ptrs->lo_d, ptrs->hi_d, U_d);
109                 /* </ex_sedov_subroutine_GPU_action_kernel.json> */
110             }
111             /* </ex_sedov_subroutine_GPU_action_loop.json> */
112         } else {
113             throw std::logic_error("[subroutine_id1_GPU] Data location is not CC1 or CC2");
114         }
115     } // OpenACC data block
116     #pragma acc wait(queue_h)
117     /* </ex_sedov_subroutine_GPU_main.json> */
118 }
119 /* </ex_sedov_subroutine_Default_main.json> */
```

Listing 14: Sedov example: Generated C++ code of GPU subroutine.

```
1  /* <ex_sedov_subroutine_Default_main.json> */
2  #include "Eos.h"
3  #include "Hydro.h"
4  #include "Driver.h"
5  #include "Flash.h"
6  /* <ex_sedov_subroutine_CPU_main.json> */
7  #include "Tile.h"
8  /* </ex_sedov_subroutine_CPU_main.json> */
9  void subroutine_id3_CPU(const int dataItem, orchestration::DataItem* dataItem)
10 {
11     using namespace orchestration;
12     /* <ex_sedov_subroutine_CPU_main.json> */
13     Tile*  tileDesc = dynamic_cast<Tile*>(dataItem);
14     /* <ex_sedov_subroutine_CPU_action_kernel.json> */
15     {
16         Grid& grid = Grid::instance();
17         Real  volumes_buffer[  (tileDesc->hi().I() - tileDesc->lo().I() + 1)
18                              * (tileDesc->hi().J() - tileDesc->lo().J() + 1)
19                              * (tileDesc->hi().K() - tileDesc->lo().K() + 1)];
20         grid.fillCellVolumes(tileDesc->level(), tileDesc->lo(), tileDesc->hi(), volumes_buffer);
21         const FArray3D  volumes = {volumes_buffer, tileDesc->lo(), tileDesc->hi()};
22         io_computeIntegralQuantitiesByBlock_CPU(dataItem, tileDesc->lo(), tileDesc->hi(), volumes, tileDesc
    ->data());
23     }
24     /* </ex_sedov_subroutine_CPU_action_kernel.json> */
25     /* </ex_sedov_subroutine_CPU_main.json> */
26 }
27 /* </ex_sedov_subroutine_Default_main.json> */
```

Listing 15: Sedov example: Generated C++ code of CPU subroutine.

Mathematics and Computer Science
Argonne National Laboratory
9700 South Cass Avenue, Bldg. #num
Argonne, IL 60439

www.anl.gov